EP3260: Machine Learning Over Networks
Computer Assignment 3
Due Date: March 7, 2023

Computer Assignment 3 - Training a neural network

Consider optimization problem

$$\underset{\mathbf{W}_1,\mathbf{W}_2,\mathbf{w}_3}{\text{minimize}} \frac{1}{N}\sum_{i\in[N]} \|\mathbf{w}_3\mathbf{s}(\mathbf{W}_2\mathbf{s}(\mathbf{W}_1\mathbf{x}_i) - \mathbf{y}_i\|_2^2,$$

where $\mathbf{s}(\mathbf{x}) = 1/(1+\exp(-\mathbf{x}))$. You may add your choice of regularizer. Using the "Individual household electric power consumption" and "Greenhouse Gas Observing Network" datasets, address the following questions:

(a) Try to solve this optimization task with proper choices of size of decision variables (matrix $\mathbf{W}_1$, matrix $\mathbf{W}_2$, and vector $\mathbf{w}_3$) using GD, perturbed GD, SGD, SVRG, and block coordinate descent. For the SGD method, you may use the mini-batch version.

(b) Compare these solvers in terms complexity of hyper-parameter tunning, convergence time, convergence rate (in terms of # outer-loop iterations), and memory requirement

Group 3: Would be great if you could add the pseudo-codes. Especially when your code does not have the implementation for PGD, SGD, SVRG and BCD.

- Adding regularizer:

$$\underset{w_1, w_2, w_3}{\text{minimize}} \quad \frac{1}{N}\sum_{i\in[N]} \| w_3 s(W_2 s(W_1 x_i)) - y_i\|_2^2 + \underbrace{\lambda\left(\|W_1\|_2^2 + \|W_2\|_2^2 + \|w_3\|_2^2\right)}$$

(b) According to the figures:

Group 3: What figures?

Hyper-parameter tunning:

GD, PGD, and BCD: only 1 hyper-parameter (learning rate) $\longrightarrow$ lowest complexity in terms of hyper-parameter tunning

SGD and SVRG: 2 hyper-parameters (learning rate, mini-batch size) convergence time :

GD, PGD, and BCD : slower

SGD and SVRG : faster

convergence rate :

GD, PGD, and BCD : slower

SGD and SVRG : faster

SGD and SVRG update the weight matrices W_1 and W_2 and the vector w_3 more frequently

Memory requirement :

GD, PGD, and BCD : lowest memory requirement (only weight matrices and vector are stored)

SGD and SVRG : highest memory requirement

weight matrices & vector + mini-batch of training samples (in SGD) and control variates (in SVRG),

Group 3: In the absence of the figures that you are referring to, it is difficult to see how you got these conclusions. Especially when you only implemented GD in your code.

$\text{layer-0} = X = f^{0}$

$\text{layer-1} = s(\omega_i x_i) = s(\omega_1 f^{(0)}) = f^{(1)}$

$\text{layer-2} = s(w_2 s(\omega_i x_i)) = s(w_2 f^{(1)}) = f^{(2)}$

$\text{layer-3} = w_3 s(w_2 s(\omega_i x_i)) = w_3 f^{(2)} = f^{(3)}$

$f = \| \underbrace{f^{(3)} - y_i}_{\text{error}} \|^2 \longrightarrow \nabla f = 2 \text{ error } \nabla f^{3}$

$\nabla f^{(3)}_{w_3} = f^{(2)}$

$\boxed{\text{layer-3-delta} = 2 \times \text{error} \times f^{2}}$ (∇ with respect to w_3)

$\boxed{\text{layer-2-delta} = 2 \text{ error} \times w_3 \nabla f^{(2)}_{(w_2)}}$ w.r.t. ω_2

$\boxed{\text{layer-1-delta} = 2 \text{ error } w_3 \; w_2 \; x_i \; \nabla f^{(1)}_{\omega_1}}$ w.r.t w_1

CA3

March 8, 2023

[1]:
```
# Import libraries
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import itertools
import argparse
import sys
import time
from sklearn import preprocessing
import pandas as pd
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
```

[2]:
```
## Preprocessing of data
# Function to load data

def get_power_data():
    """
    Read the Individual household electric power consumption dataset
    """

    # Assume that the dataset is located on folder "data"
    data = pd.read_csv('household_power_consumption.txt',
                       sep=';', low_memory=False)

    # Drop some non-predictive variables
    data = data.drop(columns=['Date', 'Time'], axis=1)

    #print(data.head())

    # Replace missing values
    data = data.replace('?', np.nan)

    # Drop NA
    data = data.dropna(axis=0)

    # Normalize
    standard_scaler = preprocessing.StandardScaler()
```

```python
    np_scaled = standard_scaler.fit_transform(data)
    data = pd.DataFrame(np_scaled)

    # Goal variable assumed to be the first
    X = data.values[:, 1:].astype('float32')
    y = data.values[:, 0].astype('float32')

    # Create categorical y for binary classification with balanced classes
    y = np.sign(y+0.46)

    # Split train and test data here: (X_train, Y_train, X_test, Y_test)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
    no_class = 2                 #binary classification

    return X_train, X_test, y_train, y_test, no_class
```

[3]:
```python
X_train, X_test, y_train, y_test, no_class = get_power_data()
print("X,y types: {} {}".format(type(X_train), type(y_train)))
print("X size {}".format(X_train.shape))
print("Y size {}".format(y_train.shape))

# Create a binary variable from one of the columns.
# You can use this OR not

idx = y_train >= 0
notidx = y_train < 0
y_train[idx] = 1
y_train[notidx] = -1
```

```
X,y types: <class 'numpy.ndarray'> <class 'numpy.ndarray'>
X size (1536960, 6)
Y size (1536960,)
```

[4]:
```python
# Sigmoid function
def sigmoid(x, derivative=False):
    sigm = 1. / (1. + np.exp(-x))
    if derivative:
        return sigm * (1. - sigm)
    return sigm

# Define weights initialization
def initialize_w(N, d):
    return 2*np.random.random((N,d)) - 1

# Fill in feed forward propagation
def feed_forward_propagation(X, y, w_1, w_2, w_3, lmbda):
    # Fill in
    layer_0 = X
```

```
    layer_1 = sigmoid(np.dot(layer_0, w_1), derivative=False)
    layer_2 = sigmoid(np.dot(layer_1, w_2), derivative=False)
    layer_3 = np.dot(layer_2, w_3)
    return layer_0, layer_1, layer_2, layer_3

# Fill in backpropagation
def back_propagation(y, w_1, w_2, w_3, layer_0, layer_1, layer_2, layer_3):
    error = layer_3 - y

    # derivative of the error with respect to w_3
    layer_3_delta = np.dot(layer_2.T, error)

    # derivative of the error with respect to  w_2
    d_layer_2 = np.dot(error, w_3.T) * sigmoid(layer_2, derivative=True)
    layer_2_delta = np.dot(layer_1.T, d_layer_2)

    # derivative of the error with respect to w_1
    d_layer_1 = np.dot(d_layer_2, w_2.T) * sigmoid(layer_1, derivative=True)
    layer_1_delta = np.dot(layer_0.T, d_layer_1)

    return layer_1_delta, layer_2_delta, layer_3_delta

# Cost function
def cost(X, y, w_1, w_2, w_3, lmbda):
    N, d = X.shape
    a1,a2,a3,a4 = feed_forward_propagation(X,y,w_1,w_2,w_3,lmbda)
    regularization = (lmbda) * ((np.linalg.norm(w_1)** 2) +( np.linalg.norm(w_2)␣
↪** 2) + (np.linalg.norm(w_3)** 2))
    return regularization + np.linalg.norm(a4[:,0] - y,2) ** 2 / N

# Define SGD
def SGD(X, y, w_1, w_2, w_3, lmbda, learning_rate, batch_size):
    # Complete here:

    return w_1, w_2, w_3

# Define SVRG here:
def SVRG(X, y, w_1, w_2, w_3, lmbda, learning_rate, T):
    # Complete here:

    return w_1, w_2, w_3

# Define GD here:
def GD(X, y, w_1,w_2,w_3, learning_rate, lmbda, iterations):
    # Complete here:
    for i in range(iterations):
        # Forward pass
```

Group 3: Looks like you did not solve these?

```
        layer_0, layer_1, layer_2, layer_3 = feed_forward_propagation(X, y, w_1,␣
↪w_2, w_3, lmbda)

        # Backward pass
        d_w_1, d_w_2, d_w_3 = back_propagation(y, w_1, w_2, w_3, layer_0,␣
↪layer_1, layer_2, layer_3)

        # Regularization
        d_w_1 += lmbda * w_1
        d_w_2 += lmbda * w_2
        d_w_3 += lmbda * w_3

        # Update weights
        w_1 -= learning_rate * d_w_1
        w_2 -= learning_rate * d_w_2
        w_3 -= learning_rate * d_w_3


    return w_1, w_2, w_3

# Define projected GD here:
def PGD(X, y, w_1,w_2,w_3, learning_rate, lmbda, iterations, noise):
    # Complete here:

    return w_1, w_2, w_3

# Define BCD here:
def BCD(X, y, w_1,w_2,w_3, learning_rate, lmbda, iterations):
    # Complete here:

    return w_1, w_2, w_3
```

Group 3: Implementation missing

[5]:
```
y_train
```

[5]:
```
array([ 1., -1., -1., ..., -1., -1.,  1.], dtype=float32)
```

[ ]:
```
# Should be a hyperparameter that you tune, not an argument - Fill in the values
parser = argparse.ArgumentParser()
parser.add_argument('--lambda', type=float, default=0.01, dest='lmbda')
parser.add_argument('--w_size', type=int, default=32, dest='w_size')
parser.add_argument('--lr', type=float, default=0.01)
parser.add_argument('--iterations', type=int, default=100)


#args = parser.parse_args()
args, unknown_args = parser.parse_known_args()
```

```
# Initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)

w_2 = initialize_w(args.w_size,args.w_size)

w_3 = initialize_w(args.w_size, 1)

# Get iterations
iterations = args.iterations
# Define plotting variables
fig, ax = plt.subplots(2, 1, figsize=(16, 8))

# Define the optimizers for the loop
optimizers = [

    {# Fill in the hyperparameters
            "opt": GD(
                X_train, y_train, w_1, w_2, w_3, learning_rate=args.lr,
                lmbda=args.lmbda, iterations=iterations),
            "name": "GD",
            "inner": None
        },
]
```

[1]:
```
# Should be a hyperparameter that you tune, not an argument - Fill in the values
parser = argparse.ArgumentParser()
parser.add_argument('--lambda', type=float, default=, dest='lmbda')
parser.add_argument('--w_size', type=int, default=, dest='w_size')
parser.add_argument('--lr', type=float, default=)
parser.add_argument('--iterations', type=int, default=)

args = parser.parse_args()

# Initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)

w_2 = initialize_w(args.w_size,args.w_size)

w_3 = initialize_w(args.w_size, 1)

# Get iterations
iterations = args.iterations
# Define plotting variables
fig, ax = plt.subplots(2, 1, figsize=(16, 8))

# Define the optimizers for the loop
```

```
optimizers = [
        {# Fill in the hyperparameters
            "opt": SGD(X_train, y_train, w_1, w_2, w_3, args.lmbda, args.lr,␣
↪batch_size),
            "name": "SGD",
            "inner": # Fill in
        },
        {# Fill in the hyperparameters
            "opt": SVRG(X_train, y_train, w_1, w_2, w_3, args.lmbda, args.lr),
            "name": "SVRG",
            "inner": # Fill in
        },
        {# Fill in the hyperparameters
            "opt": GD(
                X_train, y_train, w_1, w_2, w_3, learning_rate=args.lr,
                lmbda=args.lmbda, iterations=iterations),
            "name": "GD",
            "inner": # Fill in
        },
        {# Fill in the hyperparameters
            "opt": PGD(
                X_train, y_train, w_1, w_2, w_3, learning_rate=args.lr,
                lmbda=args.lmbda, iterations=iterations, noise=),
            "name": "PGD",
            "inner": # Fill in
        },
        {# Fill in the hyperparameters
            "opt": BCD(
                X_train, y_train, w_1, w_2, w_3, learning_rate=args.lr,
                lmbda=args.lmbda, iterations=iterations),
            "name": "BCD",
            "inner": # Fill in
        }
    ]
```

```
  File "/var/folders/l9/nl6ry63s0m3_35qt7mrjv0fh0000gn/T/ipykernel_22324/
↪3547870537.py", line 3
    parser.add_argument('--lambda', type=float, default=, dest='lmbda')
                                                        ^
SyntaxError: invalid syntax
```

Group 3: Did you check this before you uploaded the pdf?

[ ]:
```
# Run the iterates over the algorithms above

for opt in optimizers:
    #
```

```
    # Fill in
```

[ ]:
```
# Plot results
ax[0].legend(loc="upper right")
ax[0].set_xlabel(r"Iteration", fontsize=16)
ax[0].set_ylabel("Loss", fontsize=16)
ax[0].set_title("CA3 - Training a deep neural network for the power consumption␣
 ↪Dataset")
ax[0].set_ylim(ymin=0)

ax[1].legend(loc="upper right")
ax[1].set_xlabel(r"Time [s]", fontsize=16)
ax[1].set_ylabel("Loss", fontsize=16)
ax[1].set_ylim(ymin=0)

plt.savefig("power.png")
```